LBL-10189/3 c.2
UC-32

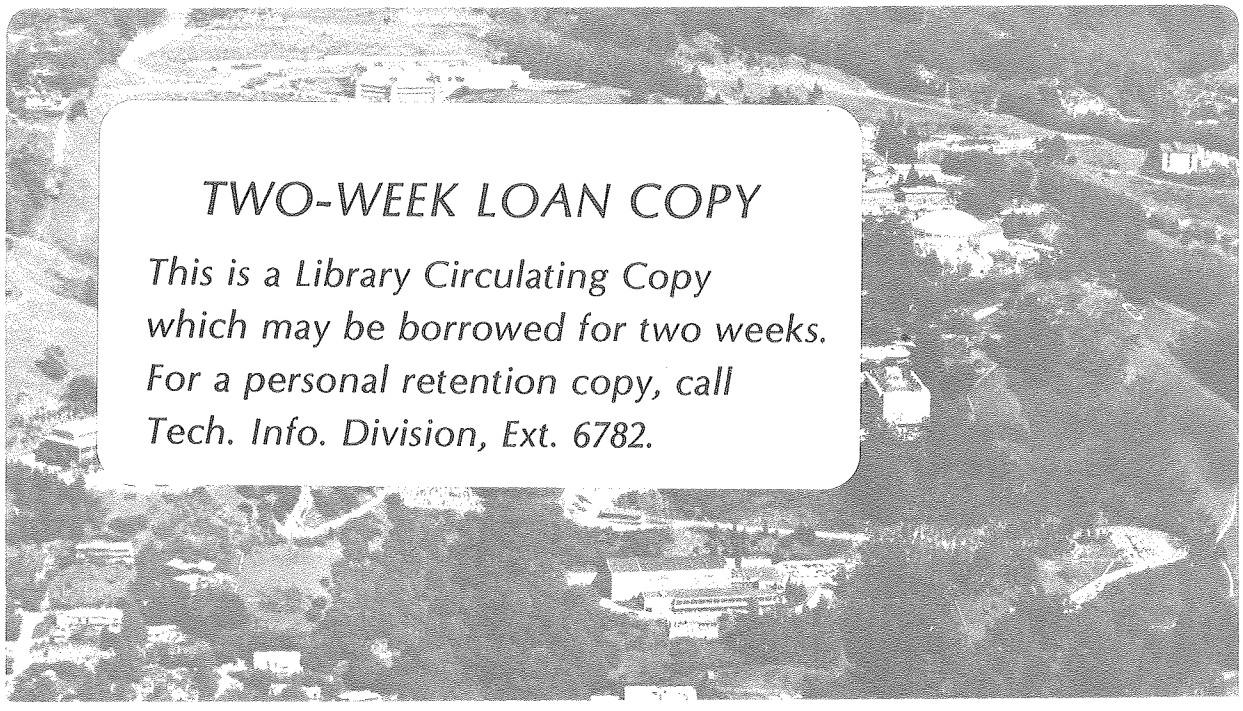
Lawrence Berkeley Laboratory
UNIVERSITY OF CALIFORNIA

Engineering & Technical Services Division

BKY PROGRAMMING SYSTEMS BULLETIN #3
1st Edition

FTN4 OPTIMIZATION TECHNIQUES

November 1979



Prepared for the U.S. Department of Energy under Contract W-7405-ENG-48

DISCLAIMER

Lawrence Berkeley Laboratory Computer Center
Systems Programming Group
University of California
Berkeley, CA. 94720

BKY Programming Systems Bulletin #3

1st Edition (Nov. 1979)
LBL-10189/3

FTN4 Optimization Techniques

by CONTROL DATA CORP.,
(Edited by R.Friedman, LBL)

CONTENTS:

This Bulletin is based on chapter 3 of the FORTRAN EXTENDED Version 4 User's Guide published by CONTROL DATA (publ. 60499700, rev. A 12/77) and edited slightly for improved clarity.

This and other Programming Systems Bulletins are available from the Computer Center Librarian, Room 50B-1245A, Lawrence Berkeley Lab., University of Calif., Berkeley, CA 94720 (415-486-5529).

0. INTRODUCTION

This bulletin describes both the optimizations that the compiler performs for the user as well as those the user can embody in the source code. Most of these optimizations decrease central processor time (not always at the expense of field length), but some decrease field length, input/output time, real time or throughput.

It should be kept in mind that the best way to optimize code is to use efficient algorithms. The higher the level at which a program is optimized, the better the results.

Array subscript computation is discussed frequently in this section; therefore, the formulas for one-, two-, and three-dimensional arrays are shown in Table 1 for convenient reference. For each typical array reference, the address calculation is shown in the form:

address of {A(subscript)} = address of A + offset * w

where offset is computed from the subscript expression as shown in the table, and w is the number of words per element defined for the array (1 for REAL and INTEGER, 2 for COMPLEX and DOUBLE PRECISION).

Because it is reasonable to assume that any programmer interested in optimal program execution will compile the program under OPT=2 or UO (unsafe optimization), the optimizations performed by the compiler in those modes are discussed first.

TABLE 1. ARRAY SUBSCRIPT FORMULAS

Number of Dimensions	Dimension Declaration	Reference Element	Offset Computation
1	A(L)	A(I)	I-1
2	A(L,M)	A(I,J)	I-1 + L*(J-1)
3	A(L,M,N)	A(I,J,K)	I-1 + L*(J-1 + M*(K-1))

1. COMPILER OPTIMIZATION

When OPT=2 is specified on the FTN control statement, the compiler optimizes the user code extensively in the process of generating object code. When the UO option is also specified, all the OPT=2 optimizations are performed, as well as some additional ones that could cause incorrect results. (The additional optimizations performed when UO is specified are identified below.)

OPT=2 mode is a global optimizer; that is, it analyzes the structure of an entire program unit during the optimization process. A brief description of the procedure followed by this optimizer will help to clarify the specific optimizations described here in more detail.

In optimizing mode, several passes are made over the source code. In the first pass, the syntax of statements is analyzed, a symbol table is constructed, and the statements are translated into an intermediate language similar to assembly language. Typically, several instructions in this intermediate language are required for each executable FORTRAN statement. At this stage, no register assignment has taken place; rather, an indefinite number of registers (R1, R2..., Rn) are used as needed. An example of a FORTRAN statement and its translation into intermediate language is shown below:

FORTRAN statement:

```
Q = X + Y/Z
```

Intermediate Language Equivalent:

```
LOAD Y --> R1
LOAD Z --> R2
DIVIDE R1 / R2 --> R3
LOAD X --> R4
ADD R3 + R4 --> R5
STORE R5 --> Q
```

The intermediate language used in this example is similar to that used by the compiler, but is different in format.

Local optimizations are performed before global optimization begins. (Local optimizations are also performed when OPT=0 or 1 is specified.) The local optimizations include constant evaluation and elimination of redundant subexpressions.

Global optimization begins by grouping sequences of intermediate language instructions into units called basic blocks. A basic block is a sequence of instructions with one entry and one point of exit. It has the property that if one instruction in a block is executed, all the instructions are executed. This grouping simplifies the process of analyzing the flow of control in the program.

In the following example, each section of code between comment lines constitutes a basic block.

```
C
        X = Y
        DO 120 I=1,N
        A(I) = B(I)
120     CONTINUE
        GO TO (100,200,300) J
C
100     Z = Q
        GO TO 500
C
200     PRINT *, X
        GO TO 500
C
300     N = N + 1
        A(I) = 0
        RETURN
C
```

The next stage is to construct a directed graph in which the basic blocks are nodes, and the lines connecting the nodes indicate a conditional or unconditional transfer between blocks. The optimizer constructs a table indicating which variables are used and defined in each block.

The optimizer then identifies all the loops in the program unit (IF loops as well as DO loops). The loops are categorized according to how deeply they are nested. (An unnested loop is in the same category as the innermost loop of a nest.) Then, beginning with the innermost loops and proceeding outward, optimizations are performed for each loop. These optimizations include movement of invariant code outside the loop, strength reduction, elimination of dead variable definitions, and register assignment.

After all loops have been optimized, object code is generated. As a result of optimization, the order in which operations are performed can be different than the order in which those operations were specified in the source code. The result, however, is always identical.

Users with a knowledge of COMPASS are encouraged to examine the object listing produced from an OPT=2 compilation to get an idea of the types of source code manipulation that take place. The listing can be compared with one produced by an OPT=0 compilation.

The compiler-produced optimizations discussed in this section are divided into machine-independent optimizations and machine-dependent optimizations. Machine-independent optimizations are those that would produce more efficient

code on any machine. Primarily, they consist of the elimination of unnecessary operations. Optimizations in this category include common subexpression squeezing, elimination of dead variable definitions, invariant code motion, and compilation time evaluation of constants. Machine-dependent optimizations are those that take into account the specific features of the systems on which FORTRAN Extended programs run. They include replacing expensive operations with cheaper ones and taking advantage of the functional units present on some models.

1.1. MACHINE-INDEPENDENT OPTIMIZATIONS

As stated above, machine-independent optimizations are those that result in the elimination of operations. In some cases, the operations are completely removed from the source code; this saves space as well as time. In other cases, operations are moved out of loops so that they are executed less frequently; this does not necessarily save any space.

Invariant Code Motion

If a sequence of instructions appears in a loop, and the result of execution of the instructions does not depend on any variable whose value changes within the loop, the instructions are called invariant. If the instructions remain in the loop, they are redundantly executed as many times as the loop is executed; therefore, the optimizer removes such sequences from loops whenever possible.

For example, in the sequence:

```
      DO 100 I=1,N
      K(I) = J/L+I**2
100   CONTINUE
```

neither J nor L can change in value during execution of the loop and, therefore, J/L is invariant and can be safely removed from the loop. J/L is then calculated only once, rather than repeatedly. After optimization, the loop is equivalent to the following:

```
      R1 = J/L
      DO 100 I=1,N
      K(I) = R1+I**2
100   CONTINUE
```

(In this example of code after optimization, and those that follow, variables of the form Rn indicate machine registers rather than memory locations; thus, the examples should not strictly speaking be read as FORTRAN statements.)

Invariant code can extend for several statements, as shown below. This example also shows that IF loops that are essentially the same as DO loops are optimized in the same way.

Before optimization:

```
100     A = P(I) + S/Q
        Y = X/Z + D
        I = I + 1
        IF(I.LT.12) GO TO 100
```

After optimization:

```
        R1 = S/Q
        Y  = X/Z + D
 100    A  = P(I) + R1
        I  = I + 1
        IF(I.LT.12) GO TO 100
```

Invariant code can also include code that is invisible in FORTRAN. For example, in the sequence:

```
      DIMENSION B(10,10,10)
      DO 10 I=1,N
      B(I,7,K) =I
10    CONTINUE
```

The relative location of the element of array B is calculated by the formula (see table 1):

```
I-1 + 10 * (6 + 10 * (K-1))
```

Without optimization, this entire calculation would be performed once for each execution of the loop. After optimization, however, the invariant part of the calculation is performed before entering the loop. This invariant part consists of the following subexpression which, in fact, is most of the calculation:

```
-1 + 10 * (6 + 10 * (K-1))
```

The optimizer only moves code out of a loop when it is certain that the code is actually invariant. There are circumstances in which execution of a sequence of instructions proves it to be invariant but the determination cannot be made at

compilation time. These circumstances include the following:

1) When a call is made to a subprogram within the loop, and the code that is being considered for invariancy uses the value of a variable that is either in COMMON or is an actual parameter to the subprogram. For example:

Example 1:

```
      COMMON X
      DO 100 I = 1,N
      A(I) = X**2
      B(I) = Y/Z
      CALL MYSUB (Q,R,Z)
100   CONTINUE
```

Neither X**2 nor Y/Z can be moved out of the loop, because the subroutine MYSUB might change the value of X or Z. However, if the call to MYSUB were:

```
CALL MYSUB (Q,R,Z+2)
```

then Y/Z could be moved out of the loop, since the compiler assumes that the call to MYSUB does not change the value of Z.

2) When a conditional branch within the loop introduces the possibility that the code might never be executed. For example:

Example 2:

```
      LOGICAL L
      DO 100 I =1,N
      IF (L) GO TO 110
      J = K+M
110   A(I) = B(I) + C(I)
100   CONTINUE
```

The expression K+M can be moved out of the loop so that it is executed only once, but the store into J must be left in the loop.

3) When the value of an expression ultimately depends on a variable that is capable of changing value in successive iterations of the loop. For

example:

Example 3:

```
      DO 100 I=1,N
      J = I+...
      K = J*...
      L = K+...
      M = L/N1 + N2*N3
100   CONTINUE
```

The division L/N1 cannot be moved out of the loop because the value of L ultimately depends on that of I, which changes each time the loop is executed.

Taking the limitations of the optimizer into account, the user concerned with optimal performance can write loops so as to maximize the amount of optimization that can take place. Above all, loop structure should be kept simple and straightforward. Common should not be used for storage of strictly local variables. Finally, expressions should be written in such a way as to make invariant subexpressions easier to recognize. For example:

```
      DO 100 I=1,N
      A(I) = (1. + X) + B(I)
100 CONTINUE
```

is preferable to

```
      DO 100 I=1,N
      A(I) = 1. + B(I) + X
100 CONTINUE
```

because 1. + X is recognized as an invariant expression only in the first case.

Common sense must be used to decide when rewriting loops interferes with the readability of code.

Whenever it is not clear whether the compiler can move invariant code, the user can move it. Moving code sometimes requires the creations of temporary variables to hold subexpressions; these variables should only be used locally, so that the optimizer does not generate unnecessary stores into them (as explained under Dead Definition Elimination). An exception to the effectiveness of this technique is that the program should not perform its own subscript calculation for a multidimensional array. For example, the sequence:

```
      DIMENSION B(10,10,10)
      DO 10 I=1,N
      B(I,7,K) = I
10    CONTINUE
```

should not be written as:

```
      DIMENSION B(10,10,10)
      ITEMP = -1 + 10*(6 + 10*(K-1))
      DO 10 I=1,N
      B(I+ITEMP) = I
10    CONTINUE
```

even though the results are the same, because the rewritten version inhibits certain special-case optimizations the optimizer performs on array subscripts. (The expression in the rewritten version is not recognized as a subscript.)

Common Subexpression Elimination

A common subexpression is an expression that occurs more than once in the source code. In completely unoptimized code, the expression is evaluated each time it occurs. Instead, the optimizer tries to save the result of the expression in a register whenever possible and to use that result instead of reevaluating the expression.

For example, in the following sequence of code:

```
X = A*B*C
S(A*B) = (A*B)/C
```

all three occurrences of A*B are matched; A*B is evaluated only once, and the result is used three times. This procedure can take place only when all of the following conditions are true:

1) The expression can be recognized as the same expression. The compiler reorders each expression into a canonical order, and then compares expressions term-by-term. Only expressions that match exactly are used. For example, A+B, A+B+C, C+D, and so forth, are recognized as subexpressions of A+B+C+D, but A+C is not recognized. B+A can be matched with A+B, however, because they are rearranged into the same order. When a subexpression contains more than one operator of equal precedence, as in:

   A*B/C

the expression is usually evaluated from left to right. Since the operators are associative, however, the compiler might reorder the operations. Parentheses can be used to ensure the desired grouping of subexpressions:

```
(A*B)/C
```

2) The expressions must be in the same basic block of code, otherwise it is not feasible to allocate a register to save the result. The further apart two occurrences of the same expression are, the less likely it is that they will be matched. Furthermore, no code can occur between occurrences of the same expression that might cause it to change in value. For example, in the sequence:

```
X = A(2)/B(2) - Q
A(I) = 4.5
Z = A(2)/B(2) + 13.4
```

A(2)/B(2) cannot be matched as a common subexpression because of the possibility that I will be equal to 2 at execution time, changing the value of the expression. In this example, if the user is sure that I will not be equal to 2, the assignment to (I) should be placed after the assignment to Z.

Keeping these restrictions in mind, the user can write expressions so as to maximize the chance that identical expression are recognized by the optimizer. For example:

```
AA = X*A/Y
BB = X*B/Y
```

is not likely to result in subexpression elimination, but

```
AA = (X/Y)*A
BB = (X/Y)*B
```

will do so.

Dead Definition Elimination

As explained above, the optimizer divides a program unit into basic blocks as part of its analysis. In the process, it keeps track of the uses and definitions of each variable within the block. By investigating which combinations

of blocks can be branched to from a given block, the optimizer determines whether the value of a variable is needed after the block is executed. If the value is needed, the variable is referred to as live on exit, otherwise it is referred to as dead on exit. If a variable is dead on exit from a block, the last store into the variable can be eliminated, since the value of the variable will not be needed again in the program.

For example, in the program unit:

```
      SUBROUTINE A (M,V,I)
      DIMENSION V(M)
      READ *,X
      GO TO (100,200) I
100   X = X/2
      IF (M .GT. 20) GO TO 250
      STOP
200   PRINT *, X
      RETURN
250   V(M) = 25.6
      RETURN
      END
```

The store into X in the line labeled 100 is eliminated, because there is no path through the program in which X could be referenced subsequently.

Locally (that is, within a basic block), other stores of a variable can also be eliminated. For example, in the sequence:

```
X = Y + Z
A = X + B
X = X/R
```

all three of these statements must be executed whenever the first one is executed. Therefore, it is not necessary to store X after the first statement because it is almost immediately redefined. A dead definition is eliminated only if the optimizer can be certain that it is really dead. For instance, the logic of the program might be such that it is impossible to decide for certain where the last usage of the variable is. In this case, no stores can be deleted (except locally). Also, the ability of the optimizer to eliminate stores even locally is limited by the availability of registers. For example, in the sequence:

```
X = Y + Z
A(I+J,J+K,K+I) = (B(M,N)+C(N,L)**(D(L,M)/E**X)/F
X = Q/R/S/T
```

It is impossible to keep the value of X in a register throughout the execution of the second statement, so X must be stored and then subsequently loaded.

There is not much the user can do to help the optimizer eliminate dead definitions. Of course, many dead definitions result from incorrect or redundant code. For example, if the last statement to be executed in a program unit is a store into a local variable, the statement is superfluous and should be eliminated by the programmer. The best advice is to keep program logic simple and avoid unnecessary use of COMMON blocks and equivalence classes.

Constant Evaluation

At all optimization levels, the compiler attempts to evaluate as many constant subexpressions as possible. The reason for this is that programs are usually executed many more times than they are compiled. For example:

```
X = 3.5 + 4.**2
```

The compiler evaluates the expression and replaces it with the constant 19.5. Some constant subexpressions serve no useful purpose and should be evaluated by the programmer, not the compiler. Others are justified, however, when they make programs more readable. This is particularly true when one of the components of the expression is a standard constant, such as pi or e. Because the expression is evaluated at compile time at minimal expense, it is better to leave such expressions unevaluated.

The user can help the optimizer by grouping constant subexpressions within an expression. For example, it is better to write:

```
X = Y*(3.14159265358979/2.)
```

than:

```
X = 3.14159265358979*Y/2.
```

because the constant subexpression is recognized in the first case but not the second.

Test Replacement

Test replacement consists of replacing, in a loop, all or some occurrences of a variable. The control variable is especially likely to be eliminated. A variable can be eliminated if it satisfies the following conditions:

- It is not in COMMON or a formal parameter

- Its value is not required outside the loop

- At least one of its appearances in the loop is in the form of a linear function (especially as an array subscript); for example:

```
      DO 100 I = 1,N
      A(2,I) = 33.2
100   CONTINUE
```

Test replacement of I can take place in this loop, but not in the following case:

```
      DO 110 I = 1,N
      X = SQRT(FLOAT(I))
100   CONTINUE
```

In test replacement, the increment and test portions of the loop code are rewritten so that a linear function of the control variable is incremented and tested, rather than the control variable itself; for example:

```
      DO 100 I = 1,N
      A(2,I) = 2.5
100   CONTINUE
```

In this loop, test replacement causes the address of the successive elements of the array A to be used for testing and incrementing, rather than the variable I. Because the distinction is easier to see in COMPASS code, the object code generated for this loop under OPT=0 and OPT=2 is shown on the next page:

Under OPT=0:

```
        SX7     1B          Initialize loop counter
        SA7     I
   )AA  BSS     0B

        SA5     CON.        Fetch constant  2.5
        SA4     I           Compute subscript reference
        BX7     X5
        SA7     X4+A-18     Store element

        SA5     I           Increment counter
        SX7     X5+18
        SX0     X7-13B
        SA7     A5
        MI      X0,)AA      Test and loop
```

Under OPT=2

```
        SA1     CON.        Initialize X1 with constant
        SB6     A+11B       Initialize B6 with highest store address
        SB7     A           Initialize B7 with first store address

   )AA  BSS     0B
        BX7     X1          Prepare store register with constant
        SA7     B7          Store
        SB7     B7+1B       Increment store address
        GE      B6,B7,)AA   Test address limit and loop
```

When the control variable has more than one use within a loop, test replacement can still take place, but the control variable is not necessarily eliminated. However, at least one increment instruction per loop iteration is eliminated.

1.2. MACHINE-DEPENDENT OPTIMIZATION

As stated above, machine-dependent optimizations are those that take advantage of the peculiar features of the systems one which FORTRAN Extended programs can be run. They fall into three main categories:

- Those that replace slower operations by faster operations. In FORTRAN, the relative speeds of operations can be ranked as follows (slowest first):

**	Exponentiation
/	Division
*	Multiplication
+-	Addition and subtraction

- Those that reorder instructions so as to use simultaneously as many functional units as possible. These optimizations are only carried out on systems with functional units; that is, the 6600, 7600, CYBER 70 Models 64 and 76, and CYBER 170 Models 175 and 176 Computer Systems.

- Those that schedule register usage so as to minimize stores and loads. These apply to all computer systems.

Strength Reduction

Strength reduction is one instance of the replacement of expensive operations by cheaper operations. Specifically, strength reduction replaces exponentiation by multiplication, and multiplication by addition.

Some types of strength reduction are local optimizations. For example, any exponentiation by a small integer constant (less than about 12) is replaced by a series of multiplications. Exponentiation by larger integers results in a call to a FTN4 Library routine, which also users multiplication for exponentiation by any integer up to about 100.

Another example is multiplication by 2, which can be replaced by addition of the variable to itself; thus:

```
J = 2*I
```

becomes:

```
J = I+I
```

When OPT=2 is specified, strength reduction also takes place in other situations. For example, if a subscript expression is of the form:

```
n*I + m
```

where n and m are unsigned integer constants, and I is a variable that varies only linearly in the loop (such as the control variable), then the multiplication can be replaced by an addition. For example, in the loop:

```
      DO 120 I=1,100
      B(4*I + 3) = 2.5
120   CONTINUE
```

the loop is rewritten as follows:

```
      R1 = 3
      DO 120 I=1,100
      R1 = R1 + 4
      B(R1) = 2.5
120   CONTINUE
```

so that the multiplication is replaced by an addition.

Special Casing of Subscripts

In a multidimensional array, subscript computation requires one or more multiply instructions. The formula for this computation is shown in table 1. If any of the declared dimensions (except the last dimension, which is not used in a multiply) is a power of 2, the multiplication can be replaced with a shift instruction which executes more quickly. This is possible because subscript dimensions are positive numbers less than 2**17 - 1. (Shifts cannot replace multiplications of other integer variables because the results might overflow 48 bits, leading to invalid results.)

In the following example:

```
      DIMENSION A(2,4,7)
         .
         .
         .
      A(I,J,K) = 452.3
```

the subscript calculation is:

```
      I - 1 + 2 * (J-1 + 4*(K-1))
```

After optimization, both multiplications are performed by shifts instead of multiply instructions.

The replacement of multiplication by a shift also takes place when the array dimension is a sum or difference of two powers of 2. In this case, the number to be multiplied is shifted twice, and the two results are added or subtracted.

In the following example:

```
DIMENSION A(6,12,3)
    .
    .
    .
A(I,J,K) = 452.3
```

the formula for the subscript is:

```
I-1 + 6*(J-1 + 12*(K-1))
```

or

```
I + 6*J + 72*K - 79
```

Both multiplications are performed using shift and add instructions.

Another type of special casing takes place when the first subscript expression of a subscript is a constant. In this case, the constant is added to the base address of array, saving one addition each time the subscript is calculated. For example, in the following case:

```
      DIMENSION A(10,10,10)
      DO 100 I=1,N
      A(4,J,I) = I
100   CONTINUE
```

the address of the array element in the assignment statement is calculated as follows:

```
Address = Base address +
(3 + 10 * (J - 1 + 10 * (I - 1 )))
```

where the base address is the address of the first element in the array. Since the constant part of the calculation only needs to be performed once, 3 is added to the base address at compile time, effectively transforming the calculation to the following form:

```
Address = Biased base address +
     (10 * (J - 1 + 10 * (I - 1 )))
```

The same principle can be applied to the case where the two leftmost subscript expression, or all three, are constants.

Functional Unit Scheduling

The central processor in several of the computer systems on which FORTRAN Extended programs run has multiple functional units. The optimizer takes advantage of this feature whenever possible by scheduling instructions so as to use units simultaneously. This optimization is performed only when the program is compiled on a system with functional units; the compiler assumes that the program is to be executed on the same system on which it was compiled. However, performance is not degraded if the program is executed on a different system.

An important special case of functional unit scheduling is array element prefetching. Prefetching takes place when the elements of an array are used successively in a loop. With prefetching, loading of the next element to be used overlaps usage of the current element. For example:

```
      DO 100 I=1,N
      A(I) = B(I) + C(I)
100   CONTINUE
```

Without prefetching, both B(I) and C(I) would have to be loaded before being added, so either the floating add unit or the increment unit (which is responsible for loads and stores) would be idle while the other unit was in use. With prefetching, B(I+1) and C(I+1) are fetched at the same time as B(I) and C(I) are added.

The potential danger with prefetching is that the last iteration of a loop might attempt to load a nonexistent array element. In the example, B(N+1) and C(N+1) are loaded (but not used) even if the arrays only have N elements. If the array is stored near the end of the user's field length, this attempt might result in an address out-of-range (an arithmetic mode 1 error). Thus, a program that executes correctly without prefetching might abort with prefetching. For this reason, prefetching is not performed for compilations under OPT=2 unless there is no danger of exceeding field length. However, when the UO (unsafe optimization) option is specified in addition to OPT=2, the compiler can prefetch for any array, without regard for the possibility of exceeding field length. Therefore, UO should not be used unless the user is sure that field length is not exceeded.

In the example above, field length is not exceeded because the increment between prefetched elements is only one word, and at least one word is guaranteed at the end of the field length.

Register Assignment

As one of the last stages of code generation, the optimizer decides which register to use for each variable and temporary quantity in every sequence of code. As part of the process, an attempt is made to minimize the number of loads and

stores required. Whenever a program uses more quantities than there are registers, some of the quantities not immediately in use must be stored and subsequently loaded. To avoid this, the optimizer analyzes the register usage of a sequence of code and decides whether to put each quantity in an A, B, or X register.

For some quantities, no alternative is available. The value of most variable or array elements must be in an X register (which is 60 bits long), and the address of an operand to be loaded or stored must be in an A register. However, any quantity known to be less than or equal to 18 bits long can be kept in either a B register (which is 18 bits long) or an X register. These quantities include DO-loop control variables, limits, and increments, and any quantity used in array subscript calculation.

In the following example:

```
      DO 100 I=J,K,L
      A(I) =B(M,N,I)
100   CONTINUE
```

I, J, K, L, M, and N can all be legally placed B registers, because none of these quantities are allowed to exceed 18 bits.

Usually, register assignment consists of reallocating quantities from X registers to B registers, since X registers are usually scarcer than B registers, but occasionally the reverse is true. A special case of register assignment is retention of B registers across calls to basic external functions (library routines), which takes place only when the UO option is specified. Normally, all registers are saved whenever an external reference occurs, because it is impossible to determine at compile time what registers are used by the referenced function. However, when the UO option is specified, the compiler assumes that certain B registers are not used by basic external functions, and does not bother to save those registers when such functions are referenced. When the UO option is specified, the user should ensure that functions with the same names as basic external functions are not loaded at execution time, unless the functions are referred to in EXTERNAL statements or type statements that override the default type.

1.3. OPTIMIZATION EXAMPLE

A somewhat more complex example can serve to illustrate how various optimizations are combined. The example below shows a simple program unit and the code that would be generated for it when compiled with each of the following two control statements:

```
FTN,OPT=0,OL.
FTN,OPT=2,UO,OL.
```

Source Code:

```
            SUBROUTINE ASCH
            INTEGER A, B, C
            COMMON A(10,10), B(10,10), C(10,10)
               DO 100 I=2,10
               A(I+1,I-1) = B(I+1,I+1) + C(I-1,I-1)
     100       CONTINUE
            RETURN
            END
```

UNDER OPT=0:

```
     *                          LINE 4
                    SX7     2B      Initialize loop counter
                    SA7     I
       )AA          BSS     0B
     *                          LINE 5
                    SA5     I       Compute subscript references
                    SX0     13B
                    DX7     X0*X5
                    SA4     X7+B    Fetch B and C elements
                    SA3     X7+C-26B
                    IX6     X3+X4   Compute sum
                    SA6     X7+A-24B     Store result in A
     *                          LINE 6
                    SA5     I       Increment loop counter
                    SX7     X5+1B
                    SX0     X7-13B  Test for completion
                    SA7     A5
                    MI      X0,)AA  Loop
     *                          LINE 7
                    EQ      EXIT.
```

UNDER OPT=2

```
     *                          LINE 4
                    SA2     B+26B   Pre-fetch initial B and C elements
                    SA1     C
                    SB5     13B     Preset address increment register
                    SB6     A+2B    Preset initial store address
                    SB7     A+132B  Preset highest store address
     *                          LINE 4
       )AA          BSS     0B
                    IX7     X1+X2   Compute sum
                    SA2     A2+B5   Fetch next B and C elements
                    SA1     A1+B5
                    SA7     B6      Store A
                    SB6     B6+B5   Increment store address
                    GE      B7,B6,)AA  Test and loop
     *                          LINE 7
```

Only the object code generated for the executable statements is shown; a full listing of the object code would also include code to allocate data blocks and COMMON blocks, and to establish communication between program units. The COMPASS instructions shown are not explained; they should be self-evident to anyone familiar with COMPASS. The code generated under OPT=2 shows the following features:

- Test replacement (the registers B6 and B7 hold linear functions of the control variable, which does not exist within the loop)

- Strength reduction (the multiplications that would normally be used in the subscript calculation have all been replaced by additions)

- Common subexpression elimination (not well shown in this example, because the expressions I+1 and I-1 have disappeared completely)

- Register assignment (use of B registers to hold array subscripts)

- Prefetching (of elements of B and C)

The object code under OPT=2 is two words shorter than the object code under OPT=0. More significantly, the loop itself is reduced from six words to two words.

2. SOURCE CODE OPTIMIZATION

A program compiled under OPT=2 almost always runs faster than a program compiled under OPT=0, or OPT=1. The amount of improvement depends primarily on the number of loops in the program, because that is where most of the optimization under OPT=2 takes place.

In addition to the optimizations performed by the compiler, the user can rewrite the source code in such a way as to improve its performance, especially for cases that the compiler is incapable of optimizing. Time should be devoted only to optimizing loops, especially innermost loops; optimizations in straight-line code are not likely to be fruitful.

Source code optimization should not be done at the expense of other desirable features; some optimizations decrease execution time while increasing field length. (This is rarely true for compiler optimizations.) Also, many optimizations decrease the comprehensibility or ease of maintenance of a program. The added cost in programmer time often exceeds the savings in execution time.

With these cautions in mind, the user can decide which of the source code optimizations described here is worthwhile in any given application.

2.1. HELPING THE COMPILER OPTIMIZE

Probably the most important source code optimizations are those intended to maximize the optimizations the compiler can perform. May of these have already been discussed in the context of the compiler optimizations, so only a summary is necessary here.

A primary consideration is to avoid unnecessary use of COMMON blocks and equivalence classes. With variables in COMMON, every subroutine call or function reference requires the compiler to store the variable before the reference, because it cannot be determined at compile time whether the variable is used in the referenced subprogram. In particular, the practice sometimes encountered of allocating local scratch variables in unused portions of COMMON blocks to save space is very detrimental, and can actually cause space to be wasted. For example, in the following sequence:

```
      COMMON I,A(1000),B(1000)
      DO 100 I=1,1000
      A(I) = 4*B(I)
      CALL SUB1 (C,D)
100   CONTINUE
      CALL SUB2
      END
```

I is in COMMON, therefore its value must be stored before each call to SUB1 or SUB2. These two stores, 30 bits each, occupy the same amount of space as the variable. If I were not in COMMON, the stores could be eliminated, saving the same amount of space and considerably speeding execution.

Equivalence classes inhibit optmization in somewhat less obvious ways. The following example is typical:

```
DIMENSION X(100)
EQUIVALENCE (X(1),W)
    .
W = Y
PRINT *,X(I)
```

Without the EQUIVALENCE statement, the assignment statement could be eliminated because the value of W is not used again in the program. However, because W is equivalenced to X(1), and the PRINT statement might reference X(1), the assignment statement cannot be eliminated.

These cautions are not meant to discourage legitimate uses of COMMON blocks and equivalence classes. In particular, when variables are needed by more than one program unit, it is faster to pass them through COMMON than as parameters, because the code for setting up and using the parameter list is eliminated.

Another major way to help the compiler optimize is to keep program logic straightforward and simple, and to keep program units short. Of course, this also improves program readability and modularity. But the advantage to the optimizer is that it its more likely to correctly identify loops and monitor the usage of variables in different portions of the program.

It has already been mentioned that the optimizer recognizes IF loops as well as DO loops. The user should keep in mind that the more closely the IF loop resembles a DO loops, the more different kinds of optimizations are performed. (The implication of this is that a DO loop should be used whenever possible.) For example, the IF loop in the following sequence of code:

```
      I = 1
100   A(I) = B(I) + C(I)
      I = I + 1
      IF (I.LE.12) GO TO 100
```

generates code essentially identical to that generated by the following DO loop:

```
      DO 100 I=1,12
      A(I) = B(I) + C(I)
100   CONTINUE
```

and all the same optimizations are performed. However, if the loop is changed to the following algebraically identical form:

```
      I = 1
100   A(I) = B(I) + C(I)
      I = I + 1
      IF(I+5.LE.17) GO TO 100
```

some of the optimizations the compiler performed in the first case cannot be performed in the second (for example, test replacement).

2.2. LOOP RESTRUCTURING

When the user is rewriting a program to optimize the source code, special attention should be paid to the loops, because that is where most of the execution time is spent in a typical FORTRAN program. Frequently, the users can take advantage of their knowledge of the peculiarities of their own program to

rewrite loops in such a way as to reduce the total number of operations performed at execution time.

One of the best known methods of restructuring is called loop unrolling. The idea is to reduce the overhead resulting from incrementing and testing the loop control variable by reducing the number of times the loop is executed. For example, the following loop:

```
      DO 100 I=1,10000
      X(I) = Z(I)**2
100   CONTINUE
```

can be replaced by this loop:

```
      DO 100 I=1,9999,2
      X(I) = Z(I)**2
      X(I+1) = Z(I+1)**2
100   CONTINUE
```

In the second case, only half as many increment, test, and branch instructions are executed.

One disadvantage of loop unrolling is that it makes programs more difficult to understand. Carried to its logical conclusion, loops would be completely eliminated, and replaced with long sequences of assignment statements. Clearly the user who is this concerned with optimization would be better off coding in COMPASS in the first place.

A more technical limitation of unrolling arises in the case when it is not known at compile time just how often the loop will be executed. For example, if the DO statement is:

```
      DO 100 I=1,J
```

unrolling does not produce the correct results unless J is an even number (assuming that each assignment statement is unrolled into two statements).

Another way to reduce the overhead associated with loops is to combine them. For example, in the sequence:

```
      DO 100 I=1,K
      A(I) = B(I) + C(I)
100   CONTINUE
      DO 110 J=1,K
      E(J) = F(J) + G(J)
110   CONTINUE
```

the two loops can be combined into one, thus reducing by half the overhead associated with the loop:

```
      DO 100 I=1,K
      A(I) = B(I) + C(I)
      E(I) = F(I) + G(I)
100   CONTINUE
```

Combining loops is usually worthwhile; however, its usefulness as an optimizing tool is limited by the requirement that both original loops must be executed the same number of times.

2.3. MISCELLANEOUS OPTIMIZATIONS

The following is a list of miscellaneous techniques for optimization which might be found helpful under particular circumstances. They are discussed very briefly.

1. Avoid mixing modes in an expression. Each conversion from one mode to another requires extra instructions. An exception is exponentiation; integers as exponents are usually quicker than real numbers, whatever the base. When a choice among modes is possible for an expression, the choice should be made according to the following hierarchy (from most efficient to least efficient):

   Integer

   Real

   Double Precision

   Double precision is especially inefficient, and should be avoided whenever possible. A single precision floating point number has 48 significant bits (15 decimal digits), which is more than enough for most purposes.

2. If a program is only going to be loaded once but executed many times, the DATA statement is preferable to assignment statements for initialization of variables, especially large arrays.

3. The forms of conditional branch, from slowest to fastest, are as follows:

   Computed GO TO

IF statement

Assigned GO TO

Unfortunately, the assigned GO TO makes following the flow of control in a program more difficult, and also impedes the detection of logic errors during debugging; it must be used with caution. When more than four or five branches can be taken from a given point, the computed GO TO is more efficient than the IF statement.

4. More efficient code is generated if one branch of an arithmetic IF or two-branch logical IF immediately follows the IF statement. In this case, the path for this statement falls through instead of branching.

5. References to basic external functions should be consolidated whenever possible. For example:

```
A = ALOG(C) + ALOG(D)
```

is not as efficient as :

```
A = ALOG(C*D)
```

Exception: Depending on values of C and D, overflow may result computing ALOG(C*D).

6. If the executable statements in a function subprogram can be consolidated into a single assignment statement, a statement function is more efficient. Because the code for a statement function is expanded inline during compilation, the overhead associated with passing parameters, saving registers, and branching to and from the function is saved for each function reference.

7. Expressions should be factored whenever possible to reduce the number of operations required for evaluation. For example:

```
X = A*C + B*C + A*D + B*D
```

should be replaced by:

```
X = (A + B) * (C + D)
```

The first version requires four multiplications and three additions; the second requires only one multiplication and two additions.

Exception: If A is very nearly equal to -B, and C very nearly equal to -D, there could be a loss of significance in the computation.

8. Use the library subroutine MOVLEV to transfer long vectors from one part of memory to another. For single vector transfers, MOVLEV is faster than the equivalent DO loop for vectors longer than 60 elements in SCM to SCM and LCM to LCM transfers, and 20 elements in SCM to LCM and LCM to SCM transfers.

3. PROGRAMMING FOR GREATER ACCURACY

The remainder of this section presents some miscellaneous ideas designed to improve the accuracy and efficiency of mathematical programs coded in FORTRAN Extended.

3.1. SUM SMALL TO LARGE

It is better to sum from small to large than from large to small. That is, when a group of numbers is to be added together, if the numbers vary widely in magnitude, a more accurate result is achieved if the smallest numbers are added first, and then the largest, rather than the other way around.

This can best be explained by an illustration. For the sake of simplicity, assume that the computer can only maintain four decimal digits of accuracy. When two numbers are added, only the four most significant digits of the result are kept, and the remainder is truncated. If the following series of numbers is to be added:

.00001234
.00005678
.00003121
.41610000
.21320000

the true result is .62940033. If the numbers are added in pairs from largest to smallest, and all but four significant digits of each result discarded, the result is .6293. If they are added from smallest to largest, the result is .6294, which is more accurate.

The explanation of this phenomenon is that, when adding from smallest to largest, because of carrying, the cumulative total of the small numbers often has one or more significant digits within the range of the larger numbers. Adding

from largest to smallest, however, the total becomes very large immediately, and smaller numbers are ignored completely.

3.2. AVOID ILL-CONDITIONING

Because of the inherent properties of certain mathematical functions, precision is increasingly lost as the function approaches a certain value. In an effort to counteract this effect, programmers often use the double precision versions of the functions. However, this technique is drastically less efficient and often produces results that are less accurate. In many cases, better and faster results can be achieved by rewriting the referencing expressions and avoiding the double precision functions.

The problem arises for values of the argument for which the derivative of the function is very large. More precisely, when the following function:

$$g(x) = \frac{xf'(x)}{f(x)}$$

is very large, which is usually true when the derivative is very large.

For example, in the expression:

```
SQRT(1.-X**2)
```

when the value of x is very close to 1., the result of the expression tends not to be very accurate. Therefore, x is frequently declared double precision, and the expression rewritten as:

```
SNGL(DSQRT(1.-X**2)
```

However, noting that:

$$1-X^2=(1-X)(1+X)$$

The expression can be rewritten with greater accuracy as:

```
SQRT((1.+SNGL(X)) * SNGL(1.-X))
```

The amplification of relative error when the value of a function is near a certain value is a particular problem with trigonometric functions. With the tangent function, the function g(x) defined above has the value:

$$g(x) = \frac{x}{\sin(x)\cos(x)}$$

g increases without limit as x approaches any multiple of pi/2 radians. When the value of x might be in this range, the programmer frequently declares x double precision and computes the function as follows:

```
SNGL(DTAN(X))
```

However, greater accuracy and efficiency can be achieved by declaring x double precision and using the addition formula for tangents (since a double precision number is the sum of its upper and lower parts). The formula is as follows (where x_u is the upper word of the double precision number, and x_l is the lower word):

$$\tan(x_u+x_l) = \frac{\tan(x_u)+\tan(x_l)}{1-\tan(x_u)\tan(x_l)}$$

Furthermore, for any number x less than 10^7, tan (x_l) is approximately the same as x_l. Therefore the formula can be rewritten as:

$$\tan(x_u+x_l) = \frac{\tan(x_u)+x_l}{1-\tan(x_u)x_l}$$

or, in FORTRAN:

```
DOUBLE PRECISION X

XU = SNGL(X)

XL = X - XU

RESULT = (TAN(XU) + XL) / (1. - TAN(XU) * XL)
```

using no double precision arithmetic.

Similar substitutions can be made for sine and cosine, using the addition formulas and the information that $\sin(x_l)$ is approximately x_l, while $\cos(x_l)$ is

approximately 1.0.

An even better example is the exponentiation function EXP. In this case, the larger the value of x, the larger the function g(x) defined above. The addition formula in this case is as follows:

$$\exp(x_u+x_l) = \exp(x_u)+x_l\exp(x_u)$$

or, in FORTRAN:

```
      DOUBLE PRECISION X

      RESULT = EXP(SNGL(X)) + (X - SNGL(X)) * (EXP(SNGL(X)))
```

* * *

Copies of this and other Programming Systems Bulletins are available from the Computer Center Library, (50B/1245A x5529).

Bulletins published to date include:

#1 Guidelines for Converting FTN4 Programs To FTN5 and the New FORTRAN-77 Standard

#2 F45 FTN4 to FTN5 Conversion Aid Reference Guide

#4 (Preliminary) Cyber Loader Reference Guide (in preparation).

1st Edition (Nov. 1979) LBL-10189/3

This work was prepared with the support of the U. S. Department of Energy under Contract W-7405-ENG-48.

U.S.GPO:1980-698-169/F-106